

Сравнение методов ускорения Python-кода: Cython vs Numba vs PyPy

Мандронов Антон Андреевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье представлено комплексное исследование эффективности трех популярных методов ускорения вычислений в Python: Cython, Numba и PyPy. Проведен сравнительный анализ производительности на тестовых задачах вычисления чисел Фибоначчи и алгоритма быстрой сортировки. Экспериментальные результаты демонстрируют значительный прирост производительности по сравнению с интерпретатором CPython. Выявлены специфические области применения каждого инструмента: Cython показывает наилучшие результаты при статической типизации и компиляции всего модуля, Numba эффективен для векторизованных операций и научных вычислений с использованием декораторов, а PyPy обеспечивает автоматическое ускорение для чистого Python-кода без модификаций. Сформулированы практические рекомендации по выбору метода оптимизации в зависимости от характеристик проекта и требований к производительности.

Ключевые слова: производительность Python, Cython, Numba, PyPy, JIT-компиляция, статическая типизация, ускорение вычислений, сравнительный анализ.

Comparative analysis of Python acceleration methods: Cython vs Numba vs PyPy

Mandronov Anton Andreevich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article presents a comprehensive study of the efficiency of three popular methods for accelerating computations in Python: Cython, Numba, and PyPy. A comparative analysis of performance was carried out on test tasks of calculating Fibonacci numbers and the quicksort algorithm. Experimental results demonstrate a significant performance gain compared to the CPython interpreter. Specific application areas for each tool were identified: Cython shows the best results with static typing and whole-module compilation, Numba is effective for vectorized operations and scientific computations using decorators, and PyPy provides automatic acceleration for pure Python code without modifications. Practical recommendations for selecting an optimization method based on project characteristics and performance requirements are formulated.

Keywords: Python performance, Cython, Numba, PyPy, JIT compilation, static typing, computation acceleration, comparative analysis.

1 Введение

1.1 Актуальность

Python является одним из самых популярных языков программирования благодаря своей простоте, читабельности и обширной экосистеме библиотек. Однако его интерпретируемая природа и динамическая типизация часто приводят к значительным накладным расходам и низкой производительности по сравнению с компилируемыми языками, такими как C или C++. Эта проблема становится критичной в областях, требующих интенсивных вычислений: научные исследования, машинное обучение, анализ больших данных и финансовое моделирование. В ответ на это было разработано множество технологий ускорения, среди которых Cython, Numba и PyPy занимают лидирующие позиции. Сравнительный анализ их эффективности, удобства использования и областей применения остается актуальной задачей, позволяющей разработчикам выбирать оптимальный инструмент для конкретных проектных требований.

1.2 Обзор исследований

В работе М. Э. Нагорных, Г. В. Твардовского и С. А. Чернышёва проводится комплексный анализ методов ускорения выполнения Python-кода с использованием различных технологий оптимизации. Исследование выявляет эффективность разных подходов к повышению производительности, включая компиляцию в промежуточный код и использование JIT-компиляторов, что предоставляет практические рекомендации по выбору оптимального инструментария для конкретных вычислительных задач [1]. В публикации А. Э. Пилюгина, А. А. Харитоновой и О. С. Дорофеевой рассматривается потенциал JIT-компилятора Numba для выполнения ресурсоемких вычислений. Подчеркивается его важность для обеспечения высокой производительности научных вычислений и обработки больших данных благодаря эффективной компиляции Python-функций в машинный код [2]. В исследовании Р. Х. Ахмедова анализируются современные технологии JIT-компиляции и оценивается их эффективность для различных сценариев использования. Рассматриваются преимущества и ограничения разных подходов к динамической компиляции, а также их влияние на производительность вычислительных систем [3]. В работе А. Ф. Зайцева проводится сравнительный анализ производительности интерпретаторов CPython и SBCL, где особое внимание уделяется оценке эффективности выполнения вычислительных операций. Исследование выявляет ключевые факторы, влияющие на производительность интерпретаторов, и предлагает рекомендации по оптимизации времени выполнения программ [4].

1.3 Цель исследования

Цель исследования — провести комплексное сравнение эффективности методов ускорения Python-кода (Cython, Numba, PyPy) на вычислительно нагруженных задачах, чтобы выявить сильные и слабые стороны каждого подхода и сформулировать практические рекомендации по их применению.

2 Материалы и методы

В рамках исследования проведен сравнительный анализ трех методов ускорения: Cython (статический компилятор и язык программирования, являющийся надмножеством Python), Numba (JIT-компилятор, использующий декораторы для компиляции функций в машинный код) и PyPy (альтернативный интерпретатор Python с JIT-компиляцией). Основное внимание уделено их влиянию на время выполнения задач.

В качестве тестовых задач выбраны:

- Вычисление n -го числа Фибоначчи (рекурсивная реализация) — задача с высокими накладными расходами на рекурсивные вызовы.
- Алгоритм быстрой сортировки (Quicksort) — алгоритм с временной сложностью $O(n \cdot \log n)$, интенсивно использующий операции сравнения и перемещения в памяти.

Разработка и тестирование проводились на Python 3.12. Для CPython, Cython и Numba использовалась среда PyCharm. Cython-код компилировался в C-расширение. PyPy тестировался в отдельном виртуальном окружении.

3 Результаты и обсуждения

Запишем в файл requirements.txt (рис. 1) версии Cython и Numba, которые будут использоваться в проекте.

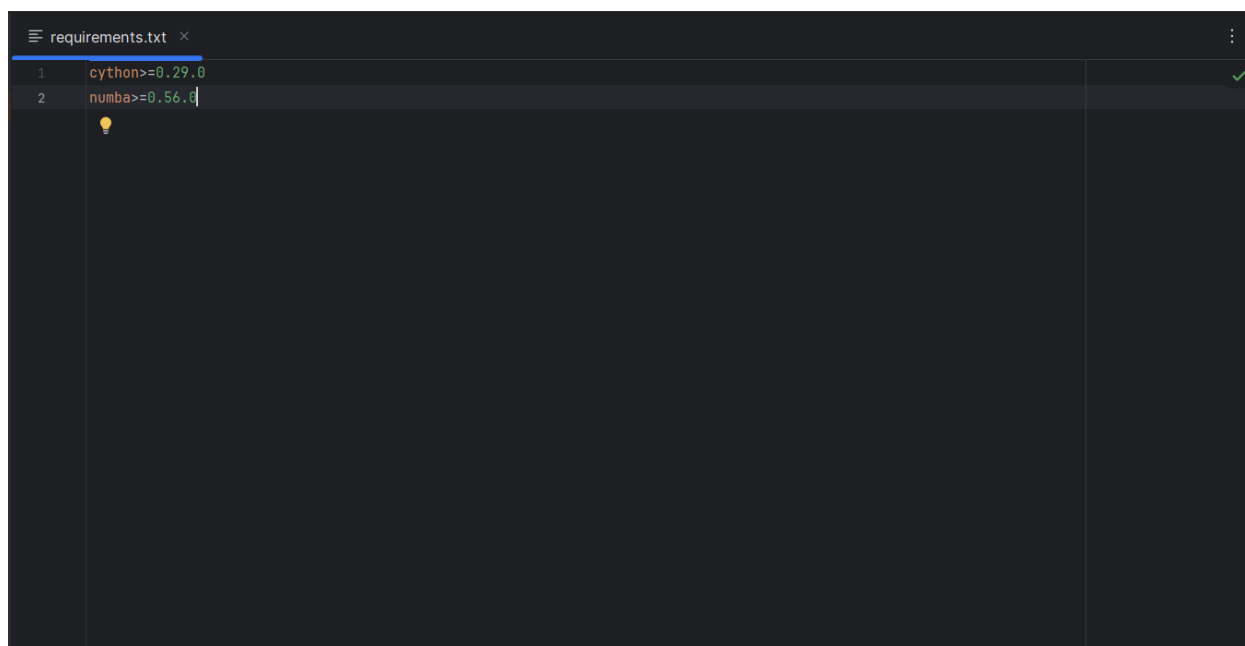


Рис 1. Requirements.txt

Установим интерпретатор PyPy через официальный сайт pypy.org (рис. 2).

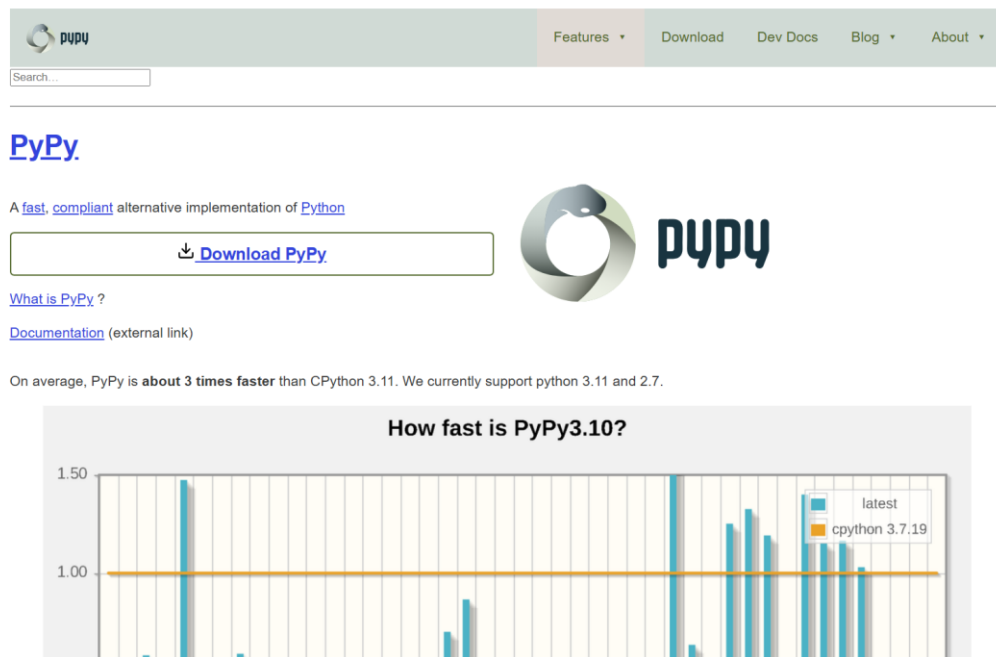


Рис 2. [PyPy.org](http://pypy.org)

Реализуем на CPython функционал вычисления n -го числа Фибоначчи в файле `fib.py` (рис. 3) и алгоритм быстрой сортировки в `quicksort.py` (рис. 4)

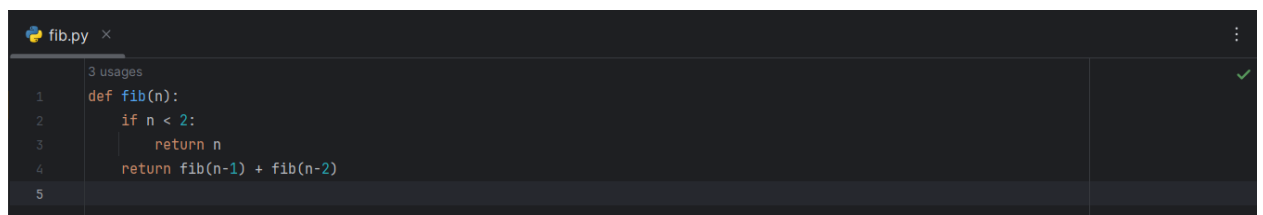


Рис 3. `fib.py`

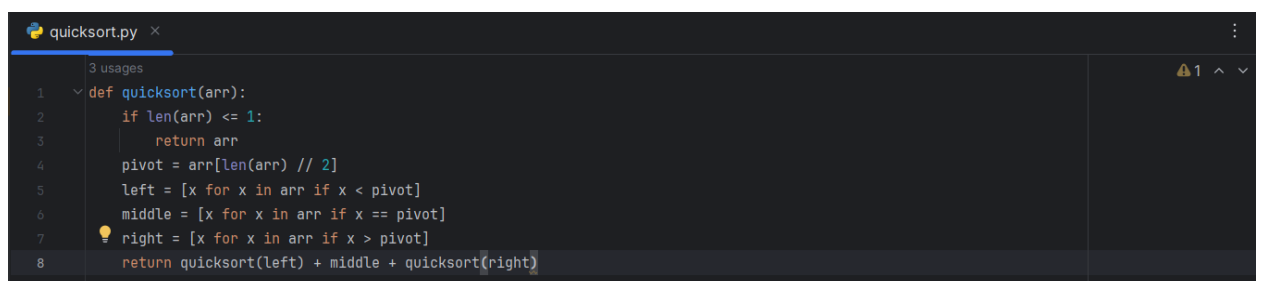
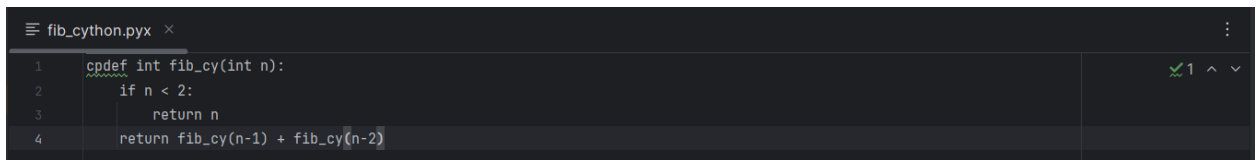


Рис 4. `quicksort.py`

Для компилятора Cython создадим файлы `fib_cython.pyx` (рис. 5) и `quicksort_cython.pyx` (рис. 6) и напомним код.

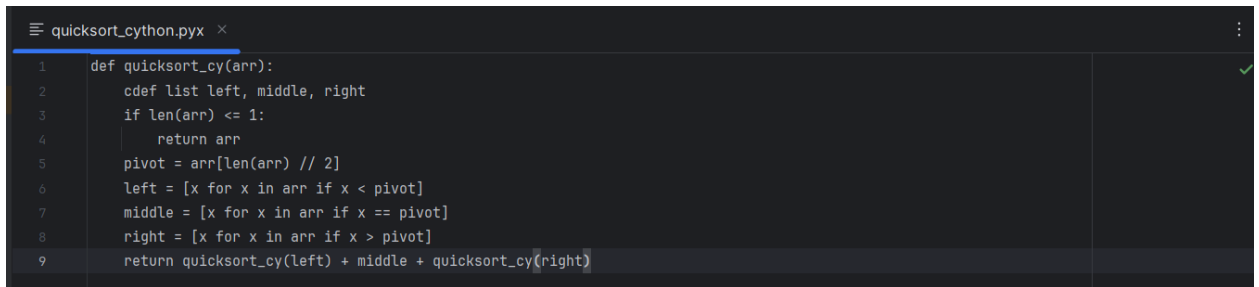


```

1  cdef int fib_cy(int n):
2      if n < 2:
3          return n
4      return fib_cy(n-1) + fib_cy(n-2)

```

Рис 5. fib_cython.pyx



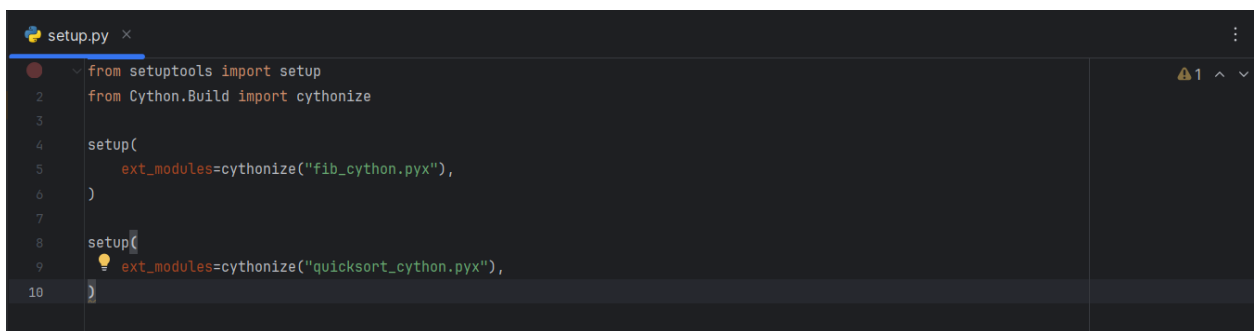
```

1  def quicksort_cy(arr):
2      cdef list left, middle, right
3      if len(arr) <= 1:
4          return arr
5      pivot = arr[len(arr) // 2]
6      left = [x for x in arr if x < pivot]
7      middle = [x for x in arr if x == pivot]
8      right = [x for x in arr if x > pivot]
9      return quicksort_cy(left) + middle + quicksort_cy(right)

```

Рис 6. quicksort_cython.pyx

Чтобы скомпилировать файлы .pyx, необходимо преобразовать код в расширение C. Для этого напомним код в setup.py (рис. 7), который переведёт написанный код в файлах на язык C.



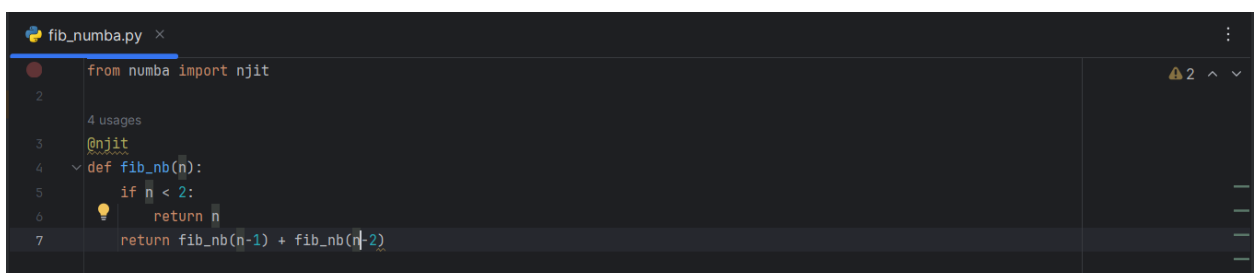
```

1  from setuptools import setup
2  from Cython.Build import cythonize
3
4  setup(
5      ext_modules=cythonize("fib_cython.pyx"),
6  )
7
8  setup(
9      ext_modules=cythonize("quicksort_cython.pyx"),
10 )

```

Рис 7. setup.py

Реализуем алгоритмы в файлах fib_numba.py (рис. 8) и quicksort_numba.py (рис. 9). В отличие от стандартного Python кода к каждой функции применяется декоратор njit из библиотеки numba для отладки в компиляторе Numba.

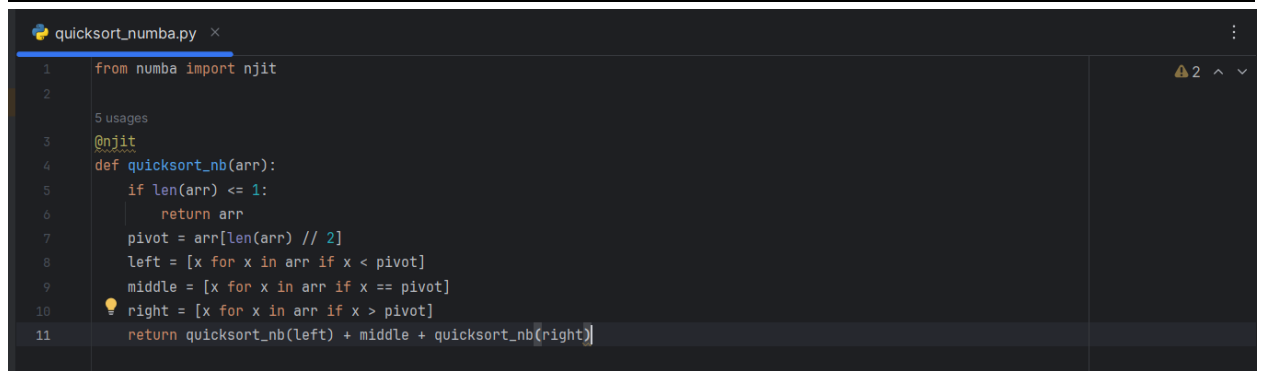


```

1  from numba import njit
2
3  @njit
4  def fib_nb(n):
5      if n < 2:
6          return n
7      return fib_nb(n-1) + fib_nb(n-2)

```

Рис 8. fib_numba.py



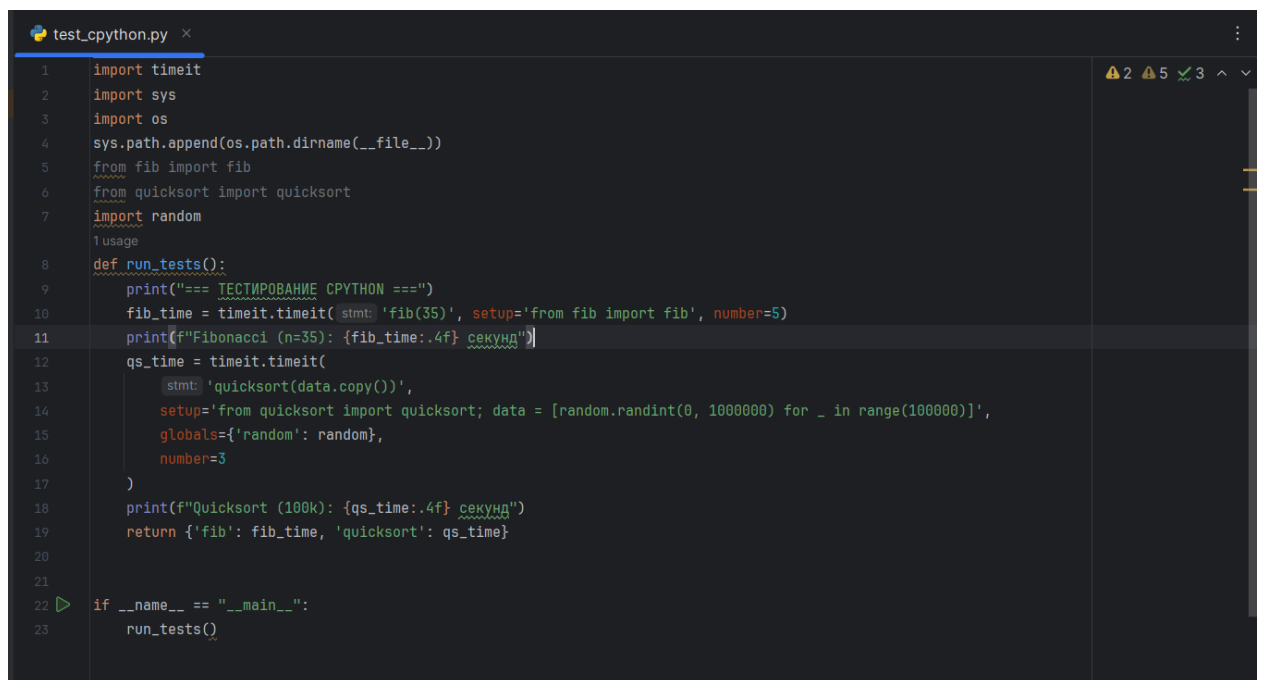
```

1  from numba import njit
2
3  @jit
4  def quicksort_nb(arr):
5      if len(arr) <= 1:
6          return arr
7      pivot = arr[len(arr) // 2]
8      left = [x for x in arr if x < pivot]
9      middle = [x for x in arr if x == pivot]
10     right = [x for x in arr if x > pivot]
11     return quicksort_nb(left) + middle + quicksort_nb(right)

```

Рис 9. quicksort_numba.py

Теперь создадим Python файлы и напомним код для запуска тестов под каждый из компиляторов. Для CPython и PyPy – test_cpython.py (рис. 10), для Cython – test_cython.py (рис. 11), для Numba – test_numba.py (рис. 12).

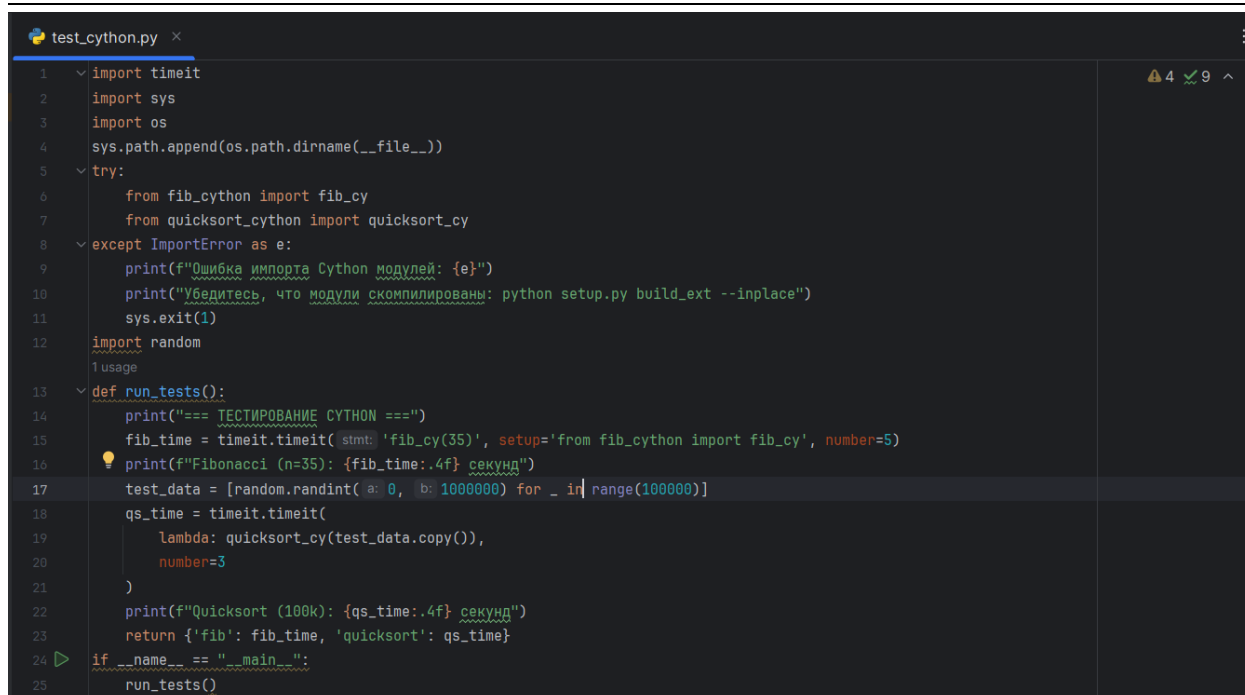


```

1  import timeit
2  import sys
3  import os
4  sys.path.append(os.path.dirname(__file__))
5  from fib import fib
6  from quicksort import quicksort
7  import random
8
9  def run_tests():
10     print("=== ТЕСТИРОВАНИЕ CPYTHON ===")
11     fib_time = timeit.timeit(stmt='fib(35)', setup='from fib import fib', number=5)
12     print(f"Fibonacci (n=35): {fib_time:.4f} секунд")
13     qs_time = timeit.timeit(
14         stmt='quicksort(data.copy())',
15         setup='from quicksort import quicksort; data = [random.randint(0, 1000000) for _ in range(100000)]',
16         globals={'random': random},
17         number=3
18     )
19     print(f"Quicksort (100k): {qs_time:.4f} секунд")
20     return {'fib': fib_time, 'quicksort': qs_time}
21
22 if __name__ == "__main__":
23     run_tests()

```

Рис 10. test_cpython.py



```

test_cython.py x
1  import timeit
2  import sys
3  import os
4  sys.path.append(os.path.dirname(__file__))
5  try:
6      from fib_cython import fib_cy
7      from quicksort_cython import quicksort_cy
8  except ImportError as e:
9      print(f"Ошибка импорта Cython модулей: {e}")
10     print("Убедитесь, что модули скомпилированы: python setup.py build_ext --inplace")
11     sys.exit(1)
12 import random
13 usage
14 def run_tests():
15     print("=== ТЕСТИРОВАНИЕ CYTHON ===")
16     fib_time = timeit.timeit(stmt='fib_cy(35)', setup='from fib_cython import fib_cy', number=5)
17     print(f"Fibonacci (n=35): {fib_time:.4f} секунд")
18     test_data = [random.randint(a=0, b=1000000) for _ in range(100000)]
19     qs_time = timeit.timeit(
20         lambda: quicksort_cy(test_data.copy()),
21         number=3
22     )
23     print(f"Quicksort (100k): {qs_time:.4f} секунд")
24     return {'fib': fib_time, 'quicksort': qs_time}
25 if __name__ == "__main__":
26     run_tests()

```

Рис 11. test_cython.py



```

test_numba.py x
1  import timeit
2  import sys
3  import os
4  sys.path.append(os.path.dirname(__file__))
5  from fib_numba import fib_nb
6  from quicksort_numba import quicksort_nb
7  import random
8  usage
9  def run_tests():
10     print("=== ТЕСТИРОВАНИЕ NUMBA ===")
11     fib_nb(10)
12     quicksort_nb([1, 3, 2])
13     fib_time = timeit.timeit(stmt='fib_nb(35)', setup='from fib_numba import fib_nb', number=5)
14     print(f"Fibonacci (n=35): {fib_time:.4f} секунд")
15     test_data = [random.randint(a=0, b=1000000) for _ in range(100000)]
16     qs_time = timeit.timeit(
17         lambda: quicksort_nb(test_data.copy()),
18         number=3
19     )
20     print(f"Quicksort (100k): {qs_time:.4f} секунд")
21     return {'fib': fib_time, 'quicksort': qs_time}
22 if __name__ == "__main__":
23     run_tests()

```

Рис 12. test_numba.py

Открываем терминал в Pycharm пишем следующие команды:

1. pip install -r requirements.txt (установка зависимостей)
2. python setup.py build_ext --inplace (компиляция Cython)
3. python test_cpython.py (запуск в среде CPython)
4. python test_cython.py (запуск в среде Cython)
5. python test_numba.py (запуск в среде Numba)
6. pyru3 test_cpython.py (запуск в среде PyPy)

Результаты тестов представлены в виде таблицы (табл. 1).

Таблица 1 – Время выполнения компиляторов в секундах

Алгоритм	CPython	Cython	Numba	PyPy
Fibonacci	6.1728	0.3415	0.2758	0.4925
Quicksort	0.6571	0.4286	0.6402	0.5389

4 Выводы

Проведенное исследование позволило сделать следующие выводы:

1. Эффективность методов ускорения варьируется в зависимости от типа вычислительной задачи. Для рекурсивных алгоритмов с интенсивными вычислениями (числа Фибоначчи) наилучшие результаты показали Numba (0.2758 с) и Cython (0.3415 с), тогда как для алгоритмов с активной работой с памятью (Quicksort) оптимальным решением оказались Cython (0.4286 с) и PyPy (0.5389 с).

2. Cython демонстрирует стабильно высокую производительность в обоих типах задач, обеспечивая ускорение в 18 раз для Fibonacci и в 1.5 раза для Quicksort по сравнению с CPython. Это делает его универсальным решением для проектов, где возможна модификация кода и добавление статической типизации.

3. Numba показала исключительную эффективность для вычислительных задач, превзойдя все другие методы в тесте Fibonacci с ускорением в 22 раза. Однако ее производительность в задачах с интенсивной работой с памятью оказалась сравнительно низкой, что свидетельствует о специализации данного инструмента.

4. PyPy обеспечивает значительное ускорение без модификации исходного кода, демонстрируя ускорение в 12.5 раз для Fibonacci и в 1.2 раза для Quicksort. Это делает его идеальным решением для проектов, где невозможно изменение исходного кода.

Библиографический список

1. Нагорных М. Э., Твардовский Г. В., Чернышёв С. А. Ускорение кода на python // Международный журнал информационных технологий и энергоэффективности. 2021. Т. 6. №. 3. С. 24-30.
2. Пилюгин А. Э., Харитонов А. А., Дорофеева О. С. Jit-компилятор Numba как инструмент для ресурсоемких вычислений //Вестник Пензенского государственного университета. 2024. №. 4 (48). С. 98-102.
3. Ахмедов Р. Х. Обзор технологий jit-компиляций //Инновации и инвестиции. 2023. №. 4. С. 278-280.
4. Зайцев А. Ф. Сравнительный анализ производительности интерпретаторов cpython и sbcl //Вестник Российского нового университета. Серия: Сложные системы: модели, анализ и управление. 2020. №. 3. С. 163-169.